AFRL-RI-RS-TR-2008-195
Final Technical Report
July 2008

OPTIMAL CONFIGURATION AND DEPLOYMENT OF SOFTWARE ON MULTI-CORE PROCESSING ARCHITECTURES

Lockheed Martin Corporation

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-195 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

WILLIAM McKEEVER
Project Engineer

/s/

JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUL 2008	Final	JUL 07- JAN 08

4. TITLE AND SUBTITLE

OPTIMAL CONFIGURATION AND DEPLOYMENT OF SOFTWARE ON MULTI-CORE PROCESSING ARCHITECTURES

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-07-C-0190

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Daniel G. Waddington

5d. PROJECT NUMBER
459T

5e. TASK NUMBER
LM

5f. WORK UNIT NUMBER
MC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lockheed Martin Corporation
3 Executive Campus
Cherry Hill, NJ 08002-4103

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITB
525 Brooks Rd Bldg 3 Suite E7
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RITB

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-195

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-3888 30-JUN-08

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Providing sufficient processing capacity within limited power and space constraints is a continuous challenge for military embedded systems. Next-generation war-fighting applications will place even greater demands on processing capacity than they do today. Commercial multi-core processors promise to solve this challenge by providing, in a comparable physical footprint, more processing capacity than their single-core counterparts.

Perseus is a suite of tools, developed under this contract, that allows existing x86-based software to be optimized for commodity multi-core platforms. Optimizations are made with respect to both performance and power consumption. The Perseus solution works by using dynamic binary instrumentation to both insert probes and modify deployed code, and by using genetic-algorithm based searches to determine optimal deployments within the potential design space.

15. SUBJECT TERMS
Multi-core, Software Optimization, Multi-Threaded Software, Power Optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			William McKeever
U	U	U	UU	40	19b. TELEPHONE NUMBER (*Include area code*) 315-330-2897

Contents

List of Figures

List of Tables

1. Executive Summary

Providing sufficient processing capacity within limited power and space constraints is a continuous challenge for military embedded systems. For example, we know from the Lockheed Martin Joint Strike Fighter (JSF) core engineering team that the aircraft Vehicle Management Computer (VMC) application design faces significant challenges in maximizing computational throughput within given design constraints. Next-generation war-fighting applications will place even greater demands on processing capacity than they do today. Commercial multi-core processors promise to solve this challenge by providing, in a comparable physical footprint, more processing capacity than their single-core counterparts.

Perseus is a suite of tools, developed under this contract, that allows existing x86-based software (in binary form) to be optimized for commodity multi-core platforms. Optimizations are made with respect to both performance (e.g., by avoiding undesirable cache effects) and power consumption (e.g., by modulating frequency and voltage of cores according to necessary workloads). The Perseus solution works by using dynamic binary instrumentation to both insert probes and modify deployed code, and by using genetic-algorithm based searches to determine optimal deployments within the potential design space.

The Perseus program has been aggressively executed over a 5 month period. The principal objective of the program was to demonstrate advanced technical concepts rather than to build robust prototypes that can be readily transitioned. The tools and technology developed under the contract, and collected experimental results have been delivered to AFRL under government rights use (see official contract for further qualification).

Figure 1. Project Success Summary

The summarized success for individual technology elements defined by the project are given in Figure 1. Color indicates level of success (green=absolutely meets expectations, yellow=somewhat meets expectations, red=does not meet expectations, grey=not applicable). This chart gives an assessment of a.) whether or not we consider the concept to be proved through the results, b.) whether the performance of the technology meets expectations and c.) the stability of the prototype implementation. The only expectation that was not met was the extraction of temporal memory maps from the probed binaries. The main issue here proved to be scalability. As part of the work we proposed an alternative approach to data collection based on partial processing of the data during collection.

2. Background

2.1 Multi-core Processing

In the last few years, commodity microprocessors have shifted towards multi-core as a means to improve performance. Traditional methods of increasing processing "power", such as increasing clock speeds, enhancing instruction-level parallelism and improving cache designs have reached a plateau – at least within the realms of economic viability. Multi-core processor designs combine multiple processors into a single die with shared cache and memory. Multi-core processors effectively provide symmetric multiprocessing (SMP)-like capabilities in a single processor package. In many cases the cores are simplified versions of their single-processor counterparts in order to take up less integrated circuit real estate. Perseus is designed for configurations where multi-core processors are homogeneous, that is, they are all identical.

The use of multi-core as a means to increase performance differs from other performance-enhancement techniques in that it is less transparent to the end user. Software must be explicitly design to be executed concurrently in order to reap the potential performance improvements that multi-core has to offer. Furthermore, software that is not carefully implemented can result in reduced performance, typically due to resource contention and other phenomena that occur in a physically parallel processing environment.

2.2 Cache False-sharing

Today's multi-core architectures typically have coherent L1 data and L2/L3 caches. L1 caches, allocated per-core, are coherent within the same processor. L2 caches are shared across processors. In the Intel IA64 architecture, the cache control protocol is MESI (modified, exclusive, shared, and invalid) based [1]. In the case of the Intel Xeon platform each L1 cache line is 64 bytes wide. When data in a cache line is written to, the MESI protocol "invalidates" copies of the same cache line in other L1 caches. Subsequent requests to read data from these invalidated cache lines result in the line being reloaded from the L2 cache (see Figure 2).

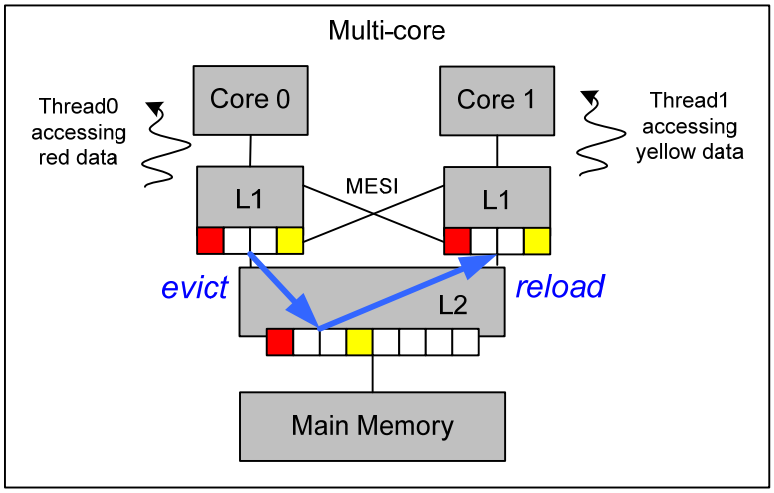

Figure 2. False Sharing

When threads running on separate cores are accessing data that resides in the same cache line (not necessarily the exact same data), the continuous evict-reload cycle causes a thrashing effect. This effect is termed "false sharing". The performance impact that false sharing brings can be significant. Benchmark tests run as part of the Perseus experimentation showed slowdowns of over 100%. The degree of slowdown directly correlates to the pattern of memory access and locality of data.

2.3 Maximizing Performance and Minimizing Power in a Multi-core Context

Many multi-core processing architectures allow the performance (with processing throughput) to be configured for each individual processing core. Performance is controlled by modulating the internal clock frequency of the core. As the clock frequency is modulated, the operating voltage of the core can also be reduced. This slow down in processing "speed" reduces the power consumption of the processing core. Power is generally proportional to clock frequency by voltage squared:

$$Power \propto Capacitance \times Voltage^2 \times Clock\ Frequency$$

Traditionally, voltage/frequency modulation technologies, such as Intel SpeedStep and AMD PowerNow, have been primarily targeted for use in mobile computing applications. Nevertheless, the same technology can also benefit embedded systems applications where power conservation is of high importance. This technology allows embedded applications to only demand the required level of processing power that is necessary at a particular time. Provided that a given task or tasks can be slowed without unduly affecting application behavior (e.g. because a task must later join with a slower task) power can be conserved. For example, the results given in Figure 3 illustrate a reduction in power consumption through slowdown. Each graph shows the power consumption over time for the same test application (which calculates Pi). Exactly the same task has been performed, yet the slower version consumes 16% less power. This benefit directly results from the reduction in frequency and voltage of the slower processor.

The notion of using "application-specific" power management [5][8] is a key element of the Perseus solution. Previous work [5] has indicated substantial power savings (over 50%) beyond static power management schemes.

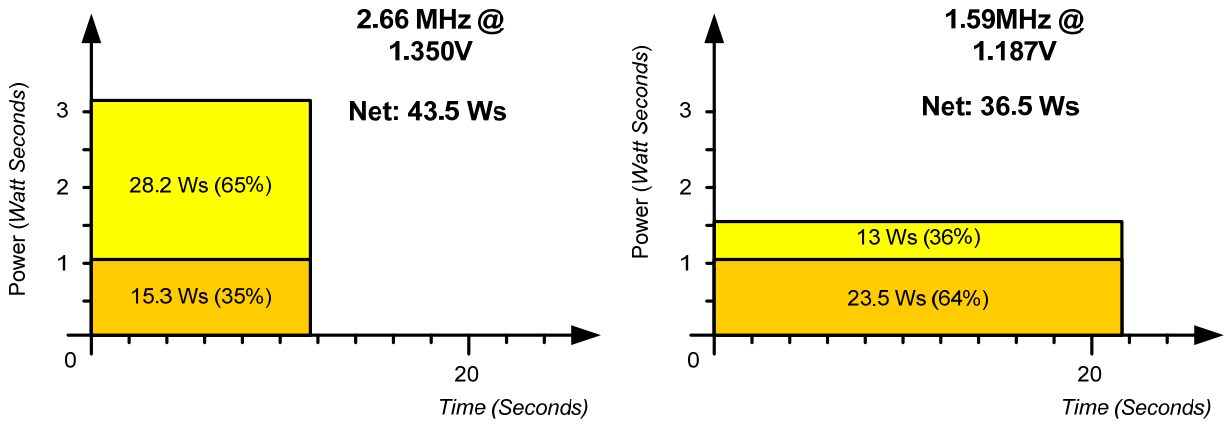

Figure 3. Results from application to calculate Pi

2.3.1 Intel Enhanced Halt State

Most Intel Xeon processors incorporate an additional power saving mechanism known as the *enhanced halt state* (feature C1E). This is a chip-triggered physical low-power state that occurs when all logical CPUs (i.e. cores) on a given processor are executing HALT or MWAIT (monitor and wait) instructions. In Figure 3, the lower sections of the graph areas illustrate the power consumed by processors in this state. The key here is that for a processor to enter the enhanced halt state *all* cores must be idle – if even one core is active the state cannot be entered. Quite often, operating system-level schedulers [6] do not take this into account in their scheduling algorithm. Consequently, scattered threads of execution can cause a system to demand more power than is necessary.

3. Problem Definition

Porting legacy applications to multi-core architectures presents two main challenges. The first problem is how, from a program design perspective, to "parallelize" the implementation so that multiple threads of control-flow can be executed concurrently. We do not address this problem in Perseus.

The second problem concerns optimizing platform[i] configuration and mapping of resources to threads of execution. The need for extensive knowledge of platform behavior and the potential complexity involved in deriving optimal designs brings the challenge outside the reach of conventional software engineering.

To efficiently and effectively optimize legacy applications for deployment to a multi-core platform, we propose the need for a set of tools that provide the following four key capabilities:

- Automatically identify "nominal" execution behavior of legacy applications that can be used as a benchmark for acceptable performance.
- Automatically identify behavior of legacy applications with respect to memory access and potential for performance degradation arising from false-sharing phenomenon.
- Automatically derive an optimal design that defines the configuration of the platform cores and mapping of threads over time (with respect to frequency/voltage modulation and availability).
- Realize the implementation of derived designs without modification of source code.

Perseus is a short-term research program (5 months) established to investigate and develop working prototype tools that can realize the above capabilities. The program scopes the investigation by demonstrating a proof-of-concept solution within the context of Linux-based systems running on Intel Xeon based multi-core processors. The investigation is also focused on repeatable applications (in a behavioral sense) that have relatively deterministic input and consistent behavior.

In formulating a plan of research, Perseus was broken down into the following technical objectives:

Table 1. Technical Objectives and Challenges

Objective	Challenges
Use binary instrumentation on x86-code to inject probes into existing software in order to extract per-thread, per-function behavioral information relating to execution periods and cycle counts.	• Collecting measurement data without affecting the normal behavior of the application. • Making modifications to binary code when complex optimizations have been used (e.g., stack arrangements).

	• Applying the technique within a Linux-based operating system and traditional user/kernel split.
Use binary instrumentation on x86-code to inject probes into existing software in order to extract per-thread behavioral information relating data and memory block access.	• Collecting accurate information when memory is shared indirectly through multiple levels of function nesting. • Collecting data at a sufficiently high rate and within a kernel/user architecture.
Identify and analyze aspects of system behavior that can be used to optimize deployment of software on to a multi-core architecture *without* modifying the intrinsic functionality of the application code.	• Determine what information can be used to drive a search-based design optimization process. • Structure design processes as a combinatorial search problem.
Use automated search algorithms to explore the design space and select optimal designs with respect to one or more performance attributes (e.g., power consumption, minimal execution time).	• Reduction of the problem. • Dealing with a large number of application and platform configuration variables that can change over time.
Construct a run-time platform and application configuration plan from the machine selected optimal design.	• Identification of control trigger insertion points and trigger code generation.
Integrated run-time platform and application configuration plan with existing x86 executable on a Linux-based system.	• Mapping of call-points to code insertion points. Writing out modified binary as a persistent solution.

The Perseus solution was designed and developed using agile methodologies. Starting with a broad definition of the system architecture, a large number of iterations were made to adapt the design to meeting technical challenges as they arose. A variation on the Scrum methodology was used to manage the research team.

4. Solution Overview

The key Perseus sub-systems are: 1) behavioral analysis – Temporal Execution Graph (TEG), 2) behavioral analysis – Temporal Memory Access Map (TMEM), 3) Design Optimization Engine, 4) Deployment Engine and 5) Run-time Engine.

Each sub-system in the solution is implemented as a separate Linux tool. In addition to the core solution we have developed a number of test applications and auxiliary tools that support experimentation (e.g., stand-alone data visualization tools).

4.1 Top-level Architecture

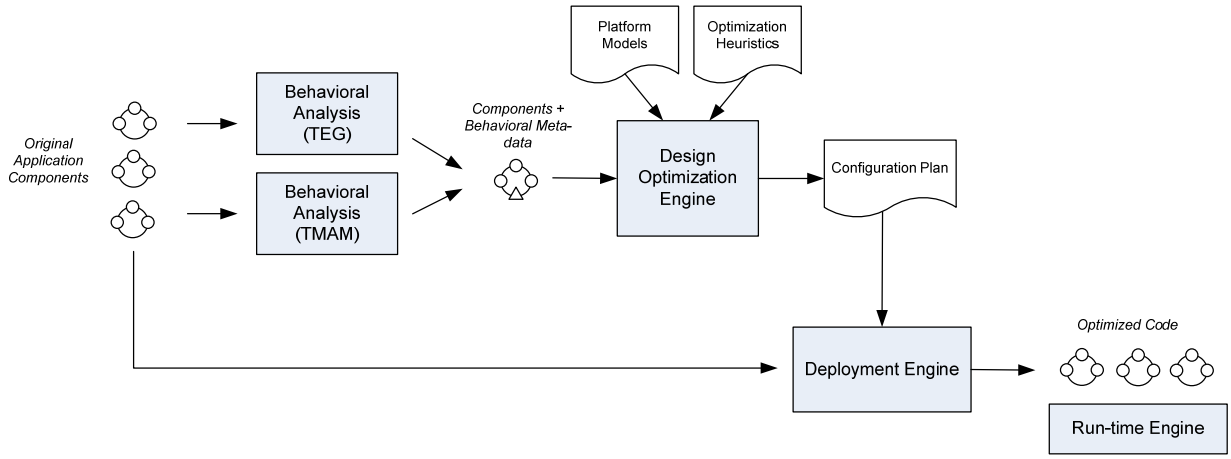

Figure 4. Top-Level Architecture

Each of the tools developed as part of the work have been tested on the Debian r.4 Linux (x86 version) running on a variety of COTS Intel platforms. Although the tools have not been officially tested on other platforms we believe that the tools can be easily migrated to support Intel x64 as well as PowerPC platforms.

Figure 5 shows the roles and relationships of the Perseus sub-systems.

Sub-system/tool	Role	Input	Output
Behavioral Analysis (TEG) `teg.exe / logger.exe`	Instrument existing binary with probes that can be used to collect per-thread, per-call site function execution times (i.e., Temporal Execution Graph). Collect auxiliary information such as shared library load site.	Original Linux Executable and Linkable Formate (ELF) 32-bit Linux Standard Base (LSB) executable.	Binary TEG data file. System map defines name to virtual address associations.
Behavioral Analysis (TMAM) `tmam.exe /`	Instrument existing binary with probes that can be used to collect	Original Linux ELF 32-bit LSB executable.	TMAM data and distilled TMAM data that defines functions

`logger.exe`	per-thread, per-instance memory block allocations and memory access patterns over time.		that are not false-sharing safe.
Design Optimization Engine `doe.exe`	Explore design-space using genetic algorithms to identify optimal configurations of the platform over time according to measured application "requirements" and platform models.	Behavioral meta-data in the form of TEG and distilled TMAM data files. Static model of the platform (e.g., # of processors, cost of frequency modulation).	Optimized dynamic configuration plan. Generated trigger code and instrumentation point specification.
Deployment Engine `deploy.exe`	Integrate dynamic configuration plan with existing source code.	Original Linux ELF 32-bit LSB executable. Optimized dynamic configuration plan.	Optimized Linux ELF 32-bit LSB executable.
Run-time Engine `fvctrl-driver.ko` `ppd.exe` `ppdcmd.exe`	Control resource partitioning. Maintain separation of Perseus-optimized applications and other applications.	None.	None.

Figure 5. Summary of Tools and their Roles

5. Detailed System Design

5.1 Behavioral Analysis Sub-system

Perseus performs behavioral analysis in two forms: temporal analysis of execution and temporal analysis of accesses to memory. The data collected by the former we refer to as the Temporal Execution Graphs (TEG). TEG data defines measurements of both cycle count and wall-clock time for execution of a given function. Distributions (i.e., for multiple executions of the function) are collected on a per-call site, per-thread basis. The latter data is referred to as the Temporal Memory Access Map (TMAM). This captures all access to memory over time on a per-thread, per-call site basis.

Both the TEG and TMAM data are ultimately leveraged by the Design Optimization Engine (refer to Section 5.5). The TEG defines "acceptable" performance with respect to time, while the TMAM is used to identify threads that exhibit cache false-sharing phenomena (see Section 2.2).

Within Perseus, data is collected by the insertion of "probes" into the existing code. Probe insertion is performed using the Dyninst [2] dynamic binary instrumentation framework. This framework allows processes to be loaded into memory and modifications made directly to the code while in memory (processes can be attached to in the same way as debuggers attach to executing processes). Dyninst allows trampoline functions to be inserted that redirect normal control flow to a newly inserted function. Trampoline functions relocate code and insert jumps into the original application. A *base trampoline* performs register saving/restoration as well as instrumentation that redirects control-flow (i.e., jumps) to one or more *mini trampolines* that call user-defined functions.

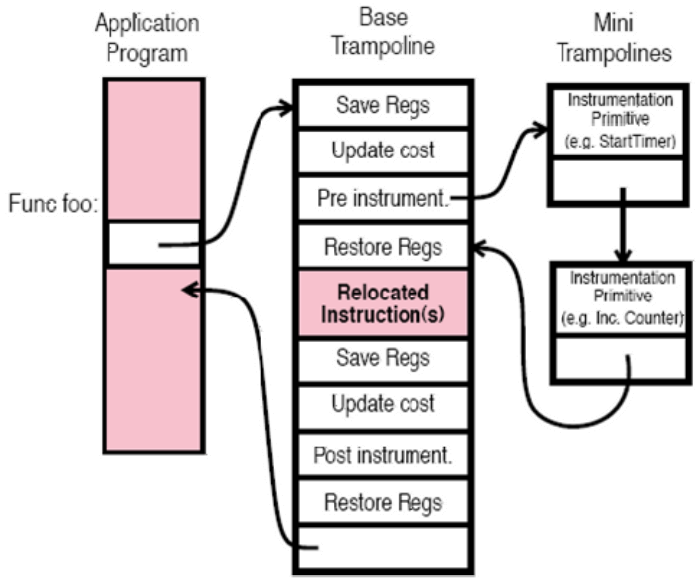

Figure 6. Dyninst Trampoline Architecture

As an alternative to Dyninst, the Pin platform [7] was also considered. However, Pin is based on the use of just-in-time compilation and thus incurs a continuous runtime overhead (which we

internally measured as exceeding 15% for our test applications). Therefore Dyninst was selected as the Perseus dynamic instrumentation technology.

5.2 High-performance Data Logger

Behavioral data is collected in the form of TEG and TMAM events. Event data is generated by the target application during execution and passed into a shared memory double-buffer. The primary sources of events are Performance Application Programming Interface (PAPI) [3] and the Intel Performance Counters [4]. These sources provide both precise timing information and chip-level information (e.g., cycles consumed).

As buffers are filled the data is written to the file system by a "collector" thread that runs on a reserved processing core. Coordination between the event generating threads and the collector thread is implemented through semaphores. The Perseus data logger is designed to minimize the impact incurred by the target application in generating, and writing out, massive volumes of event data. Both TEG and TMAM event data is stored in an uncompressed binary form to reduce disk space and minimize logger thread processing overhead. The logger uses POSIX shared memory services. Only one instance of the data logger can be run for each individual user (account) on the system.

5.3 Temporal Execution Graph Collector

TEG data is collected by inserting event probes around function calls. On triggering, each probe emits an event that identifies the thread making the call, call site address, function target address, cycle count and timestamp (refer to Appendix A). The cycle count (performance event 0x8000003B) counts the number of cycles the thread has executed not in a halt state (i.e. when not running HALT instructions). This measure essentially gives an indicator of the work done by the respective thread over time. The timestamps are read from the processor time stamp counter through the `rdtsc` (read time stamp counter) instruction. Both the cycle count and the time stamps are 64bit counters.

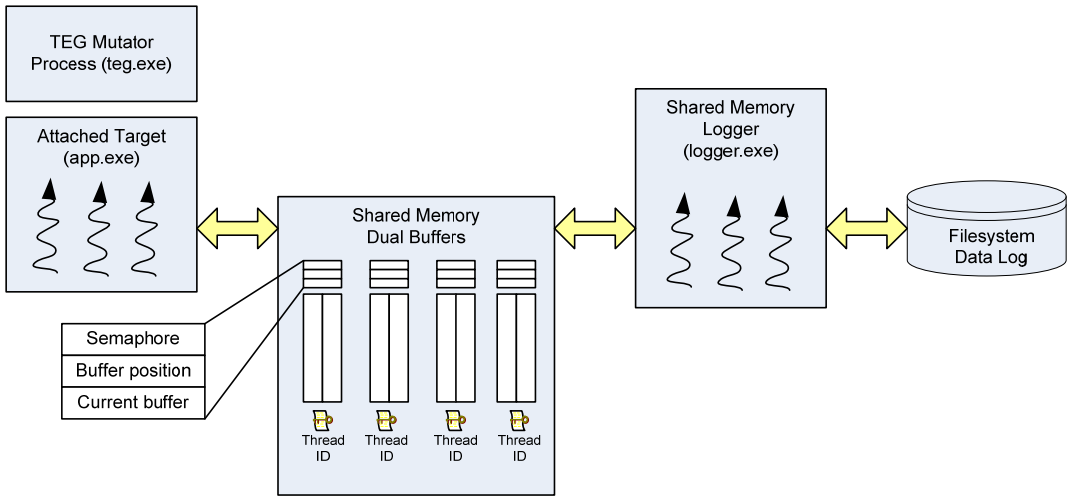

Figure 7. TEG Data Logging Architecture

As previously discussed, TEG probes pass data to the logger through shared memory. To establish a relationship with the shared logger, the TEG mutator (teg.exe) adds interceptions to the thread startup by instrumenting calls to `main` and `start_thread`. The thread startup interceptor sets up the thread's performance counters (through the PAPI API) and opens up access to the shared memory logger through a semaphore and Remote Procedure Call (RPC) buffer. The buffer information, including the current buffer and next free position, is stored in Thread Local Storage (TLS) so that it can be readily accessed as the thread runs (see Figure 7). Note that during the development of Perseus we explored the use of GCC `__thread` defined TLS. This approach was incompatible with the Dyninst framework due to modifications of the stack frame. When using TLS with PAPI, either POSIX TLS or PAPI TLS should be used.

Logging buffers are allocated on a per-thread basis – there is a one-to-one pairing of target application threads and buffering threads. As buffers are filled in the logger, the logging threads write out information to the disk. The current implementation is able to maintain in excess of one million events per second on our Intel 1.6Ghz Xeon test machine.

The TEG data collection program loads the target program into memory and modifies (i.e. mutates) the code by inserting function call interceptors at the appropriate points. All of the mutation is performed before execution of the target application begins.

5.3.1 TEG Data Visualizer

Outside of the conventional end-to-end work flow, the TEG data can be visualized with a stand alone tool. The visualizer allows one to interactively explore the TEG data and create histograms for time and cycle count distributions. Data can be filtered for specific call sites, thread identifiers, and/or function targets. The TEG visualizer is a Java program and requires the JFreeChart libraries to run. Refer to Figure 8 and Figure 9 for example visualizations.

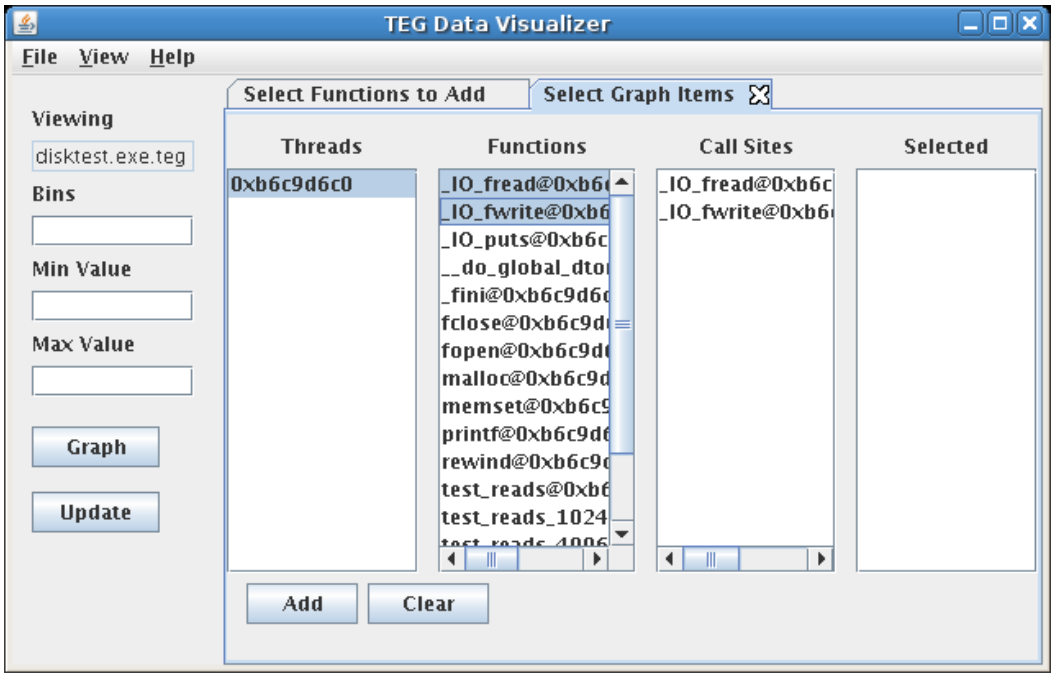

Figure 8. Data Selection In TEG Visualizer. The TEG visualizer allows the user to filter data according to thread, call site and call target. This menu shows the selection of criteria thread, function and call sites.

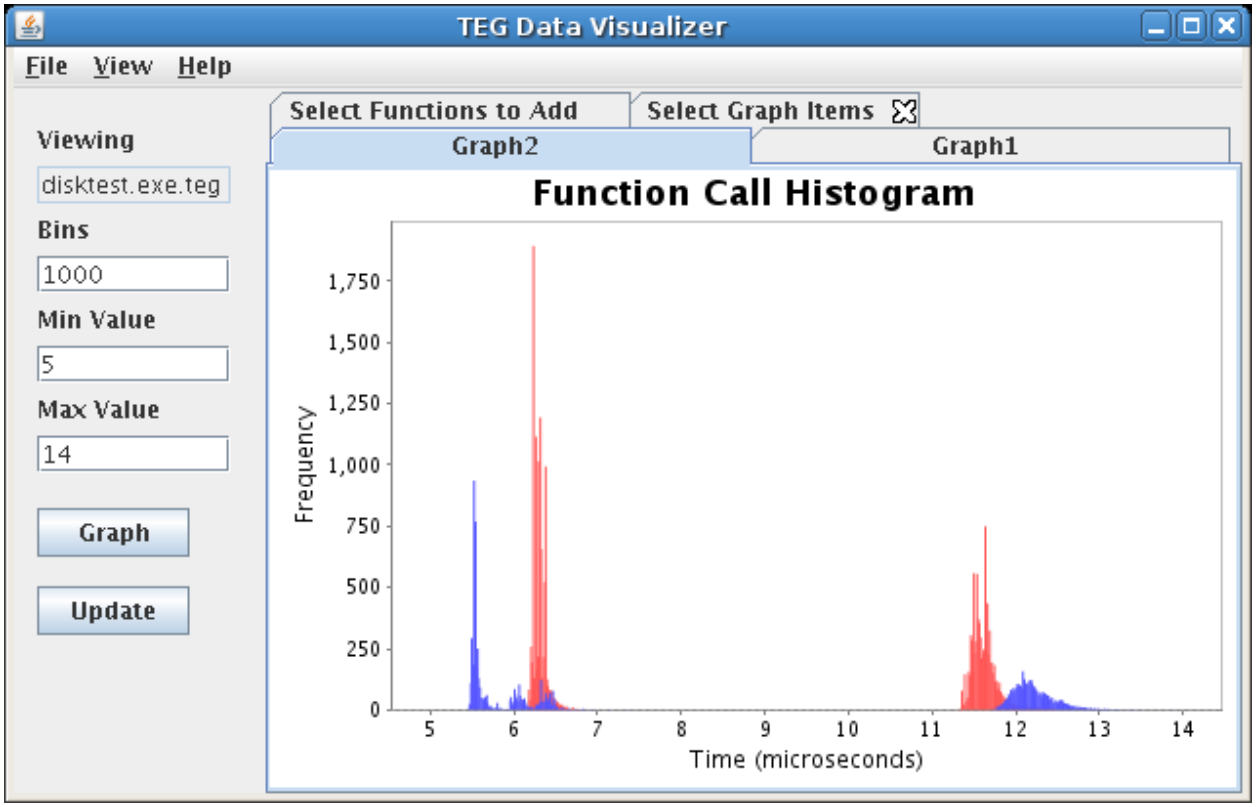

Figure 9. TEG Data Visualization. The data visualizer allows the user to build histograms of the data and make comparisons across different call site, call target and thread combinations. This screenshot illustrates how function call durations (presented in a histogram form) can be easily compared across different call sites.

Raw data and histogram data can also be exported from the visualizer, thus enabling easy visualization in other graphing applications.

5.4 Temporal Memory Access Map Collector

The Temporal Memory Access Map (TMAM) subsystem collects memory access information from program execution, in the form of reads and writes of memory, on a per-thread per-function basis. The primary purpose of this collection is to identify pairs of thread/function tuples that are likely to cause instances of false-sharing cache conflicts.

The TMAM Collector uses the Dyninst dynamic binary instrumentation toolkit, which provides an API to define an application program (the mutator) that attaches to another application program (the target) and alters the target program. In the case of TMAM, the TMAM Collector mutates the target program to identify memory accesses (loads (reads) and stores (writes)) and output information regarding each access.

A naïve approach to identifying cache conflicts records each thread/function tuple that reads or writes to a cache line and flags that set of all thread/function tuples per cache line as conflicting. This approach is very lightweight in terms of its impact on the runtime of the instrumented process. However, it is likely to have a high rate of false positives; that is, it is likely to identify pairs of thread/function tuples that do not cause false sharing conflicts.

To avoid false positives, more information must be collected and processed about each memory access. Whether the memory access was a read or a write is very important, because any number of threads can read a cache line, but a write to a cache line will invalidate that cache line in other processors' caches, potentially causing a false sharing event. Furthermore, dynamic memory management opens the possibility that two threads, the execution of which is separated in time, may access the same memory address, but that address may refer to two different allocations of memory. The TMAM must therefore collect information about memory allocations and deallocations in order to account for this. The TMAM currently collects the following tuple for each memory access:

```
(timestamp, thread ID, function address, memory block ID, access base address, access
size, type)
```

In this tuple, `timestamp` provides a total-ordering of events captured. `memory block ID` is an arbitrary integer value that uniquely identifies an allocated block of memory. `type` is one of (MALLOC, FREE, READ, WRITE, or SYNC) representing allocation events, deallocation events, reads, writes, and synchronization events. Using this rich data, the TMAM is able to output all of the information necessary to identify precisely which pairs of thread/function tuples actually cause false sharing conflicts.

The rich TMAM memory access data is output via the high-performance data logger. This is referred to as full-data mode output. However, due to the massive number of memory accesses in many programs, even this is not able to deliver acceptable performance in all cases.

We have also provided a high-speed TMAM Distiller logger that implements the naïve approach. This is referred to as distilled-data mode output. While the Distiller improves performance in many programs, it does have a higher rate of false positives.

5.4.1 TMAM Conflict Detector

The TMAM Conflict Detector tool (`tmam_conflicts`.exe) parses the output of the TMAM Collector and produces a list of pairs of thread/function tuples that cause false sharing, as well as a count of the number of observed false sharing instances per pair. The TMAM Conflict Detector operates on the TMAM full-data output only, because the distilled-data output does not contain enough information to identify conflicts with any further precision.

A simplified description of the Detector algorithm is as follows:

```
// global state
ALLOC_CL: table of cachelines containing allocated memory indexed by memory block id
TID_CL: table of cachelines indexed by the accessing thread
READ_CL: table of cachelines being read indexed by base address
WRITE_CL: table of cachelines being written indexed by base address
CONFLICTS: table of conflicts

CHECK_CONFLICTS (MSG: TMAM log message)
        Determine relevant cachelines from MSG access base address and MSG access size
        For each relevant cacheline:
                If MSG type == READ:
                        Lookup cacheline in WRITE_CL
                        If exists: Add to CONFLICTS
                Else If MSG type == WRITE:
                        Lookup cacheline in READ_CL
                        Lookup cacheline in WRITE_CL
                        If exists: Add to CONFLICTS

DEACT_CACHELINES (MSG: TMAM log message)
        Determine relevant cachelines from MSG access base address and MSG access size
        Based on MSG thread id and relevant cachelines, determine replaced cachelines
        Clear replaced cachelines from TID_CL, READ_CL, and WRITE_CL

PROCESS_ONE (MSG: TMAM log message)
        Determine relevant cachelines from MSG access base address and MSG access size
        DEACT_CACHELINES(MSG thread)
        switch(MSG type)
                case MALLOC:
                        Add relevant cachelines to ALLOC_CL
                case FREE:
                        Remove relevant cachelines from ALLOC_CL
                        Clear relevant cachelines from TID_CL
                        Clear relevant cachelines from READ_CL
                        Clear relevant cachelines from WRITE_CL
                case READ:
                        Add relevant cachelines to READ_CL
                        CHECK_CONFLICTS(MSG)
                case WRITE:
                        Add relevant cachelines to WRITE_CL
                        CHECK_CONFLICTS(MSG)

PROCESS_ALL (INPUT: TMAM log data source)
        While more messages exist in INPUT:
                Read(MSG)
                PROCESS_ONE(MSG)
```

```
Print CONFLICTS
```

The TMAM Conflict Detector operates on the (sorted) TMAM full-data output one log message at a time. When a memory block is deallocated, all of the reads and writes associated with that memory block are cleared. When a cache line is read, the Detector checks for recent writes to that cache line to look for possible conflicts. When a cache line is written, possible conflicts must be found by checking both recent reads and writes.

After examining all TMAM output, the Conflict Detector outputs the list of pairs of thread/function tuples that have caused false sharing conflicts, as well as the counts of numbers of conflicts.

5.4.2 TMAM Visualizer

The TMAM Visualizer provides a graphical interface to examine and explore memory accesses captured during an execution trace. The visualizer operates on the TMAM full-data output. The visualizer reads log messages one at a time and processes them using an algorithm very similar to that used by the Conflict Detector. Refer to Figure 10.

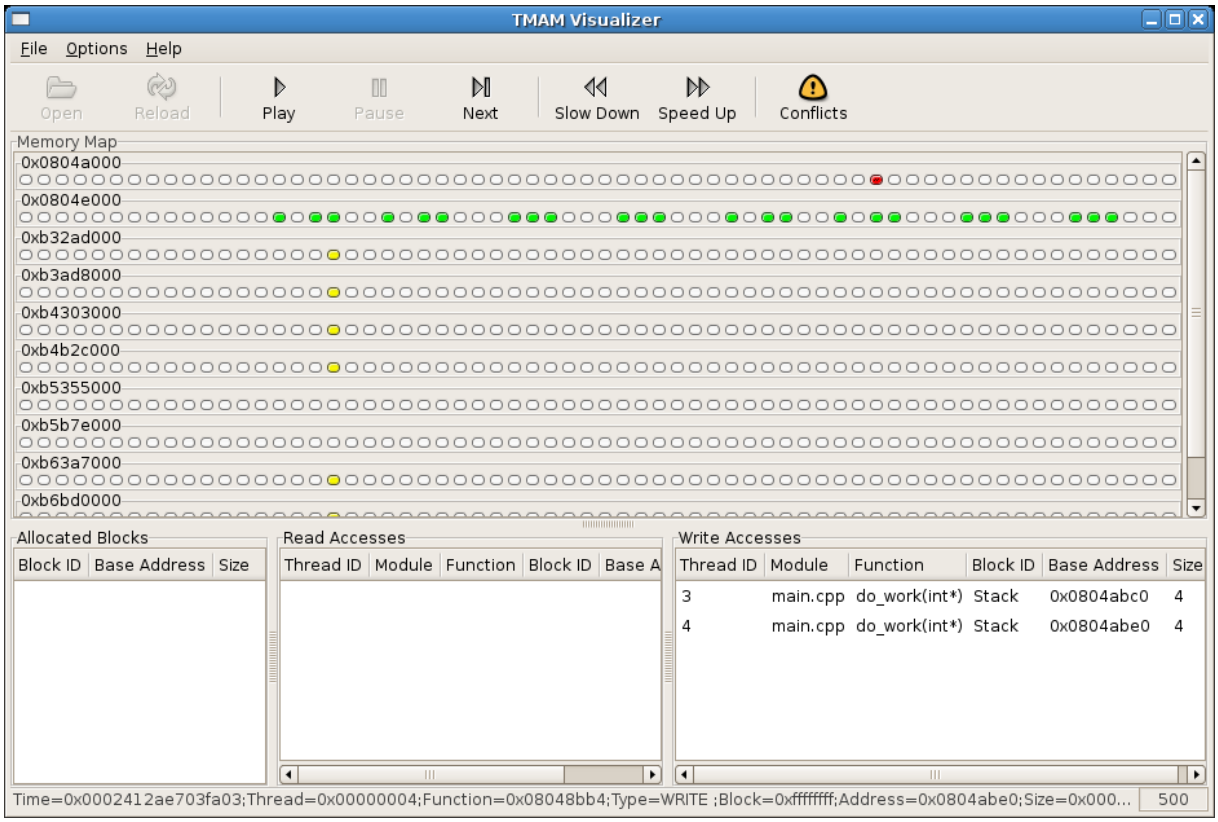

Figure 10. TMAM Data Visualization Tool. The TMAM data visualizer lets the user "play" through the event time line showing potential cache line conflicts as they occur. In the screen shot, each element of the memory map represents a single cache line of 64 bytes. Green indicates allocated memory, yellow indicates a read or write has been made to the memory and red indicates that two or more threads have made a read or write to the cache line at the same time (with respect to block allocation).

The TMAM visualizer main window interface consists of several panes. The primary pane is the memory page pane. This pane contains the space for a sorted list of memory pages in use holding data for the target program. Each page widget consists of 64 cache line widgets. Beneath the memory page pane are three panes which contain space for displaying the active allocations, reads, and writes for a selected cache line. These lists become populated during the display of a trace file.

The TMAM visualizer is called to life by opening a TMAM trace file and an associated TMAM map file. For each log message processed from the trace file, the visualizer identifies all relevant cache lines. The first time a new cache line is referenced by any log message, the visualizer identifies the memory page (aligned on a 4K boundary) to which the cache line belongs. If the page has not been referenced previously, the Visualizer creates a new widget representing the page and displays it in sorted order in the memory page pane.

Unused cache lines are colored grey. When a memory allocation (MALLOC) message is received, the relevant cache lines are colored green. When all of the memory blocks in a cache line have been freed (via FREE messages), the cache line will return to the grey color. While any cache line is being accessed (READ or WRITE messages active within that cache line) it is colored yellow. When a cache line contains conflicting accesses (READ and WRITE messages or multiple WRITE messages) it is colored red to indicate the conflict. Clicking on any cache line causes that cache line to become active in the TMAM visualizer interface, and its memory allocations, reads, and writes are displayed in the three panes at the bottom of the main window.

By default, the TMAM visualizer pauses playback on the first occasion that it identifies a given conflict, to allow the user to notice and inspect the conflict. On subsequent occurrences of the same conflict, playback is not paused. For example, if thread 1 function `Foo` conflicts with thread 2 function `Bar`, then the first such conflict (1,`Foo`,2,`Bar`) will cause execution to pause, while subsequent conflicts (1,`Foo`,2,`Bar`) will not.

In addition to this display, the visualizer also maintains a separate window, the Conflicts Window, which displays a list of all conflicts found to date.

5.5 Design Optimization Engine

The purpose of the Design Optimization Engine (DOE) is to determine a deployment plan which is tailored for the analyzed program on the platform in question. Specifically, the goal of the DOE is to determine a schedule in which threads are mapped to cores and cores are mapped to frequencies. These mappings need not be constant. For example, it is possible for a core to run at multiple frequencies during the course of an execution; similarly, threads are allowed to move from core to core. In this manner, the DOE can take advantage not only of platform specific optimizations (e.g., the Intel enhanced halt state feature) but also can adapt to changing execution patterns (e.g., by relocating threads over time to avoid false sharing).

Recall that the purpose of the TEG data is to specify required performance of the application over time. This is achieved by measuring the actual CPU cycles consumed by the application while it is active. Cumulative cycle counts are read before and after each call site. Thus, the TEG data divides the program into blocks of execution, where block boundaries are called sites.

Furthermore, each thread has its own separate cumulative count. Thus, for each thread, resource requirements are detailed at the execution block granularity. Using this information, the DOE must determine not only which core each thread should run an execution block on, but also at what frequency the core should operate at while executing the block. In essence, the DOE must search through all of the possible deployment configurations and, without executing them, determine which is optimal with respect to either execution time or power consumption (depending upon user needs). In order to accomplish this task, the DOE must utilize an extensive model of the system as well as an advanced searching technique.

5.6 System Model

In order for the DOE to determine the optimal deployment configuration, it must have some way of comparing possible deployments. In other words, the DOE must be able to abstractly execute each possible deployment against a system model in order to estimate the time and power that such a deployment would consume when actually executed. Constructing this model consisted of taking detailed measurements of the aspects of the system we considered to be the most relevant for predicting execution time and power. The quantities we measured were:

- The power consumed by each core while operating at each frequency.
- The power and time required to change the operating frequency of a core. Separate measurements were taken not only for each core, but for all pairs of frequencies (e.g., the time required to modulate core 1 from 1.8GHz to 1.6 GHz).
- The power consumed by a CPU when in the Enhanced Halt State.
- The power and time required to migrate a thread of execution from one core to another. Measurements were taken for each possible source and destination core.

In addition to these system specific costs, we also made use of the TMAM data, which described costs associated with running the application on the system. Specifically, the TMAM data lists groups of functions that when run in parallel may cause false sharing. We then associated with each false share a time penalty roughly equivalent to the cost of migrating an execution thread.

Given the execution times of all the blocks (from the TEG data) and the system model as described above, the DOE estimates the time and power for a deployment configuration by running a round robin scheduling algorithm and updating time and power costs when each block is scheduled. Of course, the actual scheduling algorithm used by the operating system is likely to be more complicated; however, we have found that in practice, round-robin yields a high quality estimation.

One issue which should be noted is that this type of simulation will only be accurate for programs which are deterministic in terms of the number of threads which are created and the work which is done by each thread. The reason for this is that the DOE uses detailed information concerning previous executions (from the TEG and TMAM data) to predict future performance; clearly if executions of a program vary drastically, the predictions are likely to be inaccurate.

5.7 Searching

Given the ability to evaluate the time or power of a possible deployment, choosing the optimal deployment is a matter of searching through the space of all possible deployments. However, considering the number of possible execution blocks of even a short running program, this space can be immensely vast. In practice, we observed that the number of possible deployment plans could easily be greater than 10^{25}. Given such a large space to search, advanced techniques need to be employed. The technique we chose was genetic algorithms; this decision was based in part upon the fact that genetic algorithms are designed to operate over large search spaces and in part because deployment plans could very naturally be represented by a gene string, the primary model used by genetic algorithms.

5.7.1 GA Background

Genetic algorithms (GA) are a searching technique based upon evolution and the principle of natural selection. In order to use a GA, a possible solution must be able to be described as a collection of genes. For example, a deployment can be viewed as a string of decisions, each of which represents a block, the associated core, and the core's operating frequency. Thus, in the same way that a person has a gene (or genes) for hair color, eye color, etc., a deployment plan has a gene for the first block to be scheduled, the second block to be scheduled, etc.

Genetic algorithms work with a collection of possible solutions called a population. Beginning with an initial generation, a GA attempts to create new solutions from existing ones, much in the same way that children are created from their parents. This is accomplished with three main operators:

- **Reproduction.** An individual is selected and a new solution is created that is a direct copy of the selected solution. In other words, a solution from the current generation is reproduced in the new generation.
- **Mutation.** A member of the current generation is selected and a slightly modified copy is added to the new generation. For example, the mutation technique we used was to change the deployment decision for a single execution block. This is an implementation of the original mutation operator discussed in [9].
- **Crossover.** Two individuals are selected and two new solutions are created by taking some gene values from each of the two parents. The two children are then added to the new generation. The specific technique we utilized was k-point crossover [10]. In this method, k cut points are made in the two parent genes and then alternating gene sequences from each parent are used to create the children. For example, if a gene string has a length of 5, a 1-point crossover might select the single cut point between gene 2 and gene 3; then each child would consist of the first 2 genes from one of the parents and the last 3 genes from the other parent. The cut points are randomly selected, thus the same two parents have the ability to create many different children.

When the GA completes, usually after a set number of generations, the fittest member of the final generation is the solution returned.

Selection is an important concept in genetic algorithms. Reproduction, mutation, and crossover all work by first selecting an individual (or two, in the case of crossover) and then performing some operation on the selectee. The manner in which individuals are selected can have a huge impact upon the algorithm. Whereas there have been many selection methods devised for genetic algorithms, all of them are probabilistically guided by the quality of the individuals; in other words, an individual solution that is better than another – or, in GA parlance, has a higher fitness – has a higher probability of being selected. This aspect of genetic algorithms models natural selection: over time, the fittest individuals persist and the less fit individuals disappear from the gene pool. For example, we chose tournament selection [11] which randomly picks n individuals and then selects the fittest of those n. A common value for n is 2. In terms of fitness, the round-robin simulator was used as the fitness evaluator; that is, if one deployment was predicted to consume less power than another, it was categorized as having a higher fitness.

5.7.2 GA Enhancements

In addition to the standard techniques described above, we also implement some well known advanced techniques in DOE genetic algorithm:

Elitism - For every generation, the $m \geq 0$ most fit individuals are automatically copied into the next generation. This technique, discussed in [10], eliminates the possibility that the best solutions are not selected for reproduction. In practice, we have found that very small values of m, such as 1 and 2, work quite well.

Introduction of Domain Specific Knowledge - Normally, the initial population is randomly selected. We have modified this slightly. In our approach, when a member of the initial population is created, there is a high probability that this individual will be randomly selected, but a small number of individuals are created intentionally from domain knowledge. Specifically, even though it is valid for the GA to switch a thread from one core to another for each execution block, in practice this is wasteful (e.g. because most of the execution time and power is spent on changing cores). How much switching constitutes “too much” is something which varies with each program and we don't know a priori what this threshold is -- in fact, finding this threshold is really what the GA is used for. However, in order to help the GA, we try to encode our limited domain knowledge that “too much changing is bad” by creating individuals that represent solutions which seldom migrate threads or not at all. We create these individuals sparingly because we don't want them to overpower the other members of the population and completely bias the GA search to one portion of the solution landscape. However, a small number of these individuals in the population can help guide the GA without completely overpowering it. We have found that a probability of 0.05- 0.1 works well.

Population Re-infusion - Since selection is based upon the fitness of individuals, it is possible for the population to converge; that is, for all individuals to be the same or very similar. When we notice that this has occurred, but the specified number of generations have not yet been created, all non-elite members of the population are discarded and replaced with new individuals. These new individuals are created in the same way as those for the initial population. Essentially, population re-infusion re-starts the GA while remembering the best individuals seen so far. We claim that a population has a low diversity when the difference between the fitness of the best individual and the fitness of the worst individual are within 10% of the fitness of the best

individual.

Incest Prevention - If two identical parents are involved in a k-point crossover, then the children created are exact duplicates of the parents. In this case, crossover simply mimics reproduction. Even if the two parents are not identical, but very similar, the children produced are nearly identical to the parents, making crossover quite like mutation. Both of these cases can cause the population to converge prematurely. In order to prevent such convergence, we ensure that children are sufficiently different from their parents by utilizing a technique called incest prevention [12]. Basically, we only allow two individuals to be involved in a crossover if they are sufficiently different from each other. We use hamming distance in order to measure the similarity of two individuals.

5.7.3 Platform Control Plan

The final design is output from the DOE as a platform control plan. Each plan consists of a set of "trigger" points (defined as memory addresses in the code) as well as a source code (C/C++) to implement the triggers. Triggers effectively implement application-driven control on the platform by setting the affinity of the current thread as well as setting the frequency/voltage of specific cores. Trigger points define where the appropriate triggers should be grafted into the original code and, as discussed in the previous section, the cost of triggers and their functionality is taken into account during the design optimization process.

5.7.4 GA Visualization

As part of the Perseus tool set, the process of search the solution space via the GA algorithm can be viewed using a GUI-based visualization tool (Figure 11).

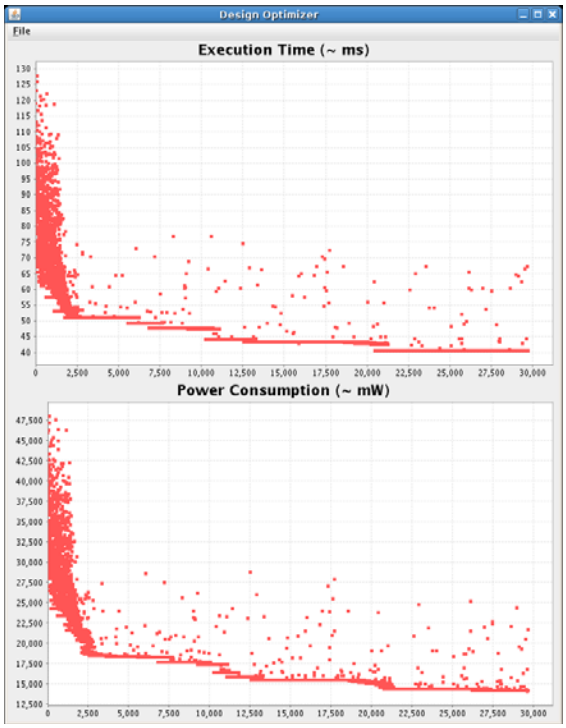

Figure 11. Real-time Visualization of Genetic-Algorithm Populations (Designs). This snapshot is of the GA visualization tool. This tools allows the tool user to visualize progress of the GA in real-time. This particular view shows the current best performance (with respect to execution time and power consumption) as the GA runs.

5.8 Deployment Engine

The deployment engine is responsible for taking the original application binary and modifying it to realize the platform control plan. This is done by instrumenting the program binary with triggers (i.e. calls) into the control plan. In the current prototype the control plan is compiled into a dynamically loadable library so that it can be loaded into the application's process space together with platform control libraries for voltage/frequency modulation and thread affinity control[ii]. See Figure 12.

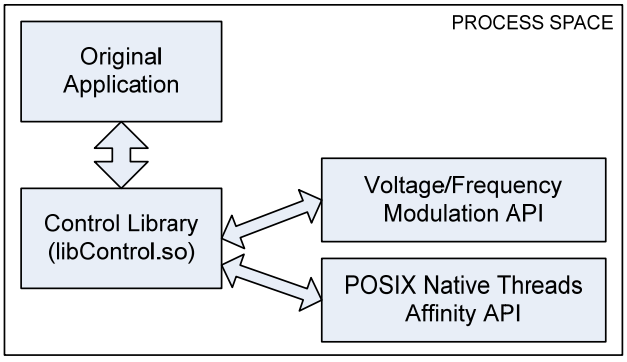

Figure 12. Final Application Deployment Structure. Final deployment of the optimized application is achieved by loading a design-specific control library into the application's process space, which is hooked into the application by inserting "triggers" at the appropriate points in the binary code.

Because the final application is instrumented, there is an overhead (on the order of 10-15 seconds) incurred for loading the process and performing the instrumentation. The experimental results do not include this overhead as we believe that in any practical deployment the application would be instrumented and the modified binary written out. Alternatively, trigger points can be hand grafted into the code and the control plan included as part of the application source. We believe that this approach would lend itself to formal certification through code reviews, a practice common to DoD systems development.

The following code illustrates an example (from the haltstate gate test) of the Perseus generated control plan:

```
#include <pthread.h>
#include "affinity.h"
#include "fvctrl.h"
#include "triggeraux.h"

void Init_Frequency()
{
        modulate_cpu(0, 0, 0);
        modulate_cpu(1, 0, 0);
        modulate_cpu(2, 0, 0);
        modulate_cpu(3, 0, 0);
        modulate_cpu(4, 0, 0);
        modulate_cpu(5, 0, 0);
        modulate_cpu(6, 1, 0);
        modulate_cpu(7, 1, 0);
}

void Set_Default_Affinities()
{
        switch(GetThreadInstanceId()){
                case 0: {
                        affinize_thread(0, pthread_self());
                        break;
                }
                case 1: {
                        affinize_thread(4, pthread_self());
                        break;
                }
                case 2: {
                        affinize_thread(5, pthread_self());
                        break;
                }
        }
}

void Before_CS_8048D26()
{
        switch(GetThreadInstanceId()) {
                case 0: {
                        affinize_thread(5, pthread_self());
                        break;
                }

        }
}

void Before_CS_8048D75()
{
```

```
        switch(GetThreadInstanceId()) {
                case 1: {
                        affinize_thread(4, pthread_self());
                        break;
                }

                case 2: {
                        affinize_thread(4, pthread_self());
                        modulate_cpu(5, 1, 0);
                        break;
                }

        }
}
```

6. Experimental Results

6.1 Test infrastructure

The results in this document were collected on the following platform:

HP Proliant DL140 Rack Mount Server S/N USE731NBHC
Dual- Quad-core Intel Xeon Processors E5320 @1.86GHz

Intel Performance Primitives 5.3.1 (I23-28101A01D)
Intel C++ Compiler Standard v10.0.023 Edition for Linux I23-35101A01D
Linux (Debian R4.1.1) Kernel version 2.6.18-10.00.Custom build (gcc version 4.1.2)

6.2 Experimental Power Measurement

Power consumption of the test platform was measured by attaching a real-time streaming digital multi-meter (Agilent 34405A Digital Multimeter S/N TW47270003) directly to the ATX motherboard power supply. This particular motherboard conforms to the server ATX specification and thus has two 12V ATX lines, one feeding each of the processors (see Figure 13).

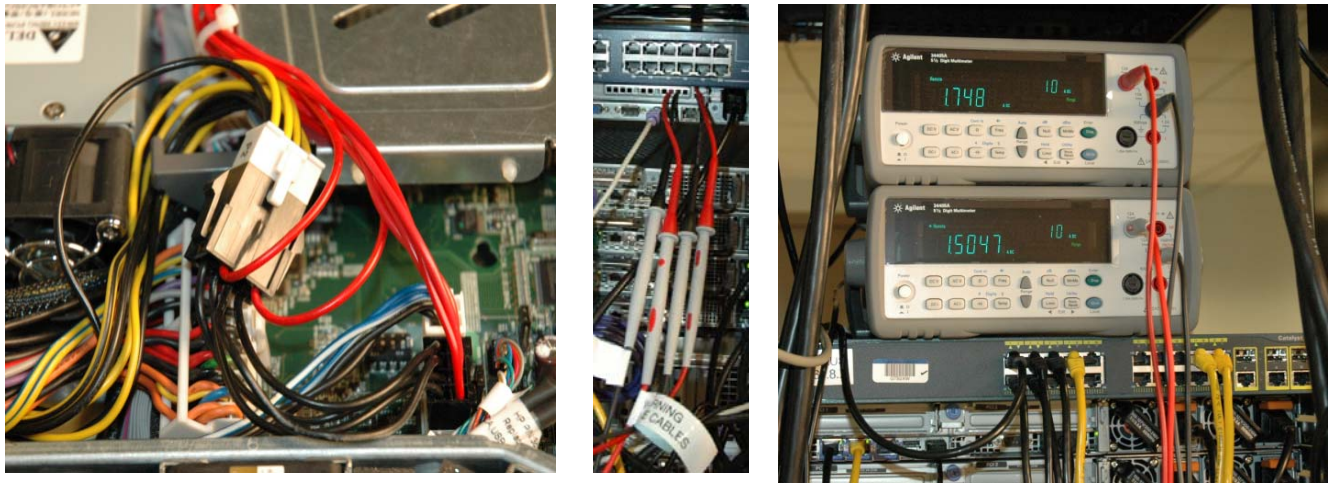

Figure 13. Perseus Power Measurement Infrastructure. Power consumption is measured directly from the ATX supply lines on the system motherboard. Two multimeters are used, one per processor.

Data is streamed from the multimeters in real-time via a USB-connected host PC. Samples are taken at a rate of approximately 2 per second. As part of the work, we developed our own data streaming and capture application that supports multiple data sources (i.e. multimeters) and the ability to 'mark' data points during capture. Data points are triggered from the host being measured via a TCP/IP connection to the host PC. Figure 14 shows the visualization provided by this application.

Figure 14. Perseus Real-time Power Measurement Tool. Data is streamed from the Agilent multimeters in real time via USB to a collection PC.

6.3 Experimental Gate Tests

In order to evaluate the technology developed during the Perseus project, we identified a number of gate tests that would demonstrate different aspects of the solution. Each gate test has been constructed with different optimizations in mind. Each has a known "optimal design" and thus as Perseus is run on these gate tests the expected outcome is understood *a priori*.

Note: The following results were taken during the Perseus program. Since completion of the AFRL funded effort, ATL has performed additional experimentation and made a number of modifications to the design optimization engine.

Table 2. Description of Experimental Gate Tests and each objective

Test #	*Name*	*Description*	*Objective*
1.	Haltstate.exe	Two counting threads and a joining main thread. Each counting thread has equal workload and priority. There is no data sharing across threads and thus false sharing is not expected.	Co-locate threads to a single CPU in order to facilitate power-reduction by entering enhanced halt-state on a single processor.
2.	Float.exe	Four counting threads plus a joining main thread. Each counting thread has equal workload and priority. There is no data sharing across threads and thus false sharing is not expected.	Reduce frequencies of core belonging to main thread and other cores not performing work.
3.	Pairs.exe	Eight counting threads plus a joining main thread. Threads are divided into four pairs. Each pair is counting data in the same cache line and thus false-sharing potential is high.	Co-locate pairs to avoid false sharing.

4.	Imgapp.exe	Application based on Intel Performance Primitives libraries. Five threads process data in a fork/join pattern. Different threads perform different types of work. Work types include high-pass, Gaussian, and Sobel-Horiz filtering. Final stage of processing operates only three threads over a period of approximately 10 seconds.	Reduce frequencies of cores running computational low-demanding filters. In final fork join pattern, co-locate remaining working threads to same processor for enhanced halt state activation.
5.	Signal.exe	Signal processing application based on Intel Performance Primitives libraries. Application applies Hamming Windows, and Discrete Cosine Transform (DCT) algorithms to matrices. Sorting stages also included. Application has six threads in phase one and two threads in phase two.	Defined as a representative application of the signal processing application domain. Phase changes should result in enhanced halt state activation.

The design optimizer was configured with platform information collected empirically through the micro-benchmarks. These benchmark tools, which can be found in the Perseus SourceForge repository (sourceforge.atl.external.lmco.com), are used to collect performance data for relevant aspects of the system. Key measurements include thread migration cost, frequency control cost and rates of instructions-to-power. Appendix A2 shows an example configuration for the `float.exe` gate test.

Table 3 and Table 4 provide results collected from the Perseus gate test experiments (see repository/testing/gatetests directory).

Table 3. Genetic-Algorithm Running times for each Gate Test

Application	Code Size (Kb)	GA Running Time	GA Iterations
float.exe	82	4m 35s	2000
haltstate.exe	15	2m 35s	1000
pairs.exe	82	2m 45s	1000
imgapp.exe	774	59m 54s	1000
signal.exe	65	4 hr	1000

Table 4. Power/Performance Changes for Optimized Applications

Application	Total Reduction in Execution Time (%)	Normal Power (Ws)	Optimized Power (Ws)	Power Saving (%)
float.exe	-1.86	145.55	115.4	20.7
haltstate.exe	2.79	54.05	41.82	22.6
pairs.exe	19.25	79.82	65.161	18.4
imgapp.exe	6.13	483.00	469.65	2.7
signal.exe[iii]	8.67	535.38	527.5	1.5

Overall the results are positive. However, by studying the data closely it is clear that further tuning of the GA is needed (resulting in the relatively small power savings achieved for the image and signal processing applications). In the tests we have seen "obvious" optimizations not being made for the image processing and signal processing applications. For example, we have seen runs where co-location and spare-core frequency reduction is clearly advantageous to power conservation. We speculate that the reason for these deficiencies is either that the platform modeling parameters are inaccurate or that the GA search is getting stuck in a local minima.

Also note that these experimental results are representative and application dependent. The running times used were relatively small (less than 5 minutes) so that data bloat did not become a problem from the perspective of the GA. In future work, we will be looking at increasing the computing power for the GA processing to scale to larger designs.

7. Project Resources

All of the Perseus project information, source code and documentation are maintained on the Lockheed Martin ATL SourceForge web site at:

https://sourceforge.atl.external.lmco.com/sf/sfmain/do/viewProject/projects.perseus

Requests to join the Perseus project should be made to *dwadding@atl.lmco.com*.

8. Conclusions

Over the five month timeframe of this program, a substantial research and development effort has ensued. The key objective of the program was to demonstrate, through a proof-of-concept prototype, that a) naïve migration of legacy applications to multi-core typically results in poor performance with respect to power and time, and that b) machine-learning could be used to effectively solve the complex temporal mapping problem of thread-to-core and core configuration.

Key Observations:

Naïve Migration Leads to Less-than-Optimal Performance – Our results have shown that there are simple optimizations that can be made by paying attention to the nuances of multi-core cache architectures (e.g., false-sharing phenomena) and power management algorithms (e.g., enhanced halt state). Our data shows gains in performance in the region of 20% and savings in power also around the 20% mark. These measurements were taken from a two-stepping platform and thus we anticipate that additional savings could be achieved on platforms with more steppings.

Effectiveness of Application-specific Power Management - State-of-the-art microprocessor-level (e.g., Intel Speedstep) and OS-level (e.g., `cpufreq` governors) power management strategies are unable to exploit application characteristics that can provide obvious savings in power. A clear example of this is the inability to co-locate threads onto processors in order to free up whole processors for shutdown or enhanced halt state. It could be argued that modifications to the operating system scheduler could help alleviate this problem but this could come at a cost to the scheduling overhead.

Data Bloat in Memory Maps - Temporal memory maps in their raw form are too large in size to manage and process. Although we demonstrated that information maintained in a temporal memory map is directly useful in avoiding phenomena such as false-sharing, collecting that map data without any form of distillation is impractical for most reasonably sized applications. From our experimentation, we now believe that memory map data must be distilled in real-time in order to avoid excessive data bloat.

Effectiveness of Binary Instrumentation - Dynamic binary instrumentation is effective in probe insertion. Nevertheless, our experience has shown that binary instrumentation technology is still in its infancy. Our experience with Dyninst is that, while being able to deal with most code, there are still a reasonably large number of compiler and system optimizations that cannot be instrumented by the current solution. Examples are the use of compiler specific thread-local storage (e.g., GNU `__thread` attribute) and stack optimizations. Furthermore, modified binaries are often brittle and susceptible to unpredictable behavior.

Application Predictability – We have also demonstrated that many applications (particularly those relevant to the DoD domain) are relatively predictable with respect to execution time. Our use of the temporal execution graph (TEG) to empirically define a datum for behavior, from which optimizations can be based, has shown to be effective. The key to the success of the approach is the appropriate selection of measurement granularity and the existence of recurring equations in the processing.

Degrees of Modulation – The experimental work for this program was performed on a dual-quad-core Intel Xeon platform with only two frequency modulation steppings. The number of

steppings available is defined as a relation of the front-bus frequency and the maximum processor clock speed. Today, the most steppings an Intel Xeon processor can support is six. We believe that this additional range of scaling would allow additional gains from the Perseus optimization.

Although the program has developed a functional prototype and a number of key findings in the area of power-optimization for multi-core, there is still a large body of research that remains. Further investigations that require attention include: a) configuration of machine-learning algorithms to drive system design and optimization processes; b) the effectiveness of greater frequency/voltage modulation levels (e.g., O(1000)) in processors and the move towards on-demand power models that eliminate the need to operate processors during halt cycles; and c) extension of the Perseus optimization strategies to the heterogeneous multi-core domain.

It is clear that new technologies bring new capabilities but also bring new challenges. The advent of multi-core has demonstrated a clear need for a change in the way we build (and parallelize) software systems – a challenge clearly at the forefront of many commercial technology vendors such as Intel, IBM and NVIDIA. Nevertheless, outside of the forward looking challenges, the defense industry is faced with issues and challenges that are not of immediate concern to the broader industry. Key examples include the problem of legacy software and the need for program code certification. Keeping abreast and engaged with these additional challenges is key to the long term success of the defense industry.

9. References

[1] H. P. Messmer, "The Indispensable Pentium Book", Addison Wesley Publishing, ISBN 0-201-87727-9, 1995.

[2] Buck, B., & Hollingsworth, J. K. "An API for Runtime Code Patching", International Journal of High Performance Computing Applications, pp. 317-329, 2000.

[3] Browne, S., Deane, C., Ho, G., & Mucci, P. "PAPI: A Portable Interface to Hardware Performance Counters", Processings of Department of Defense HPCMP Users Group Conference, June 1999.

[4] Intel Corporation. "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2", November 2006.

[5] AbouGhazaleh, N., Childers, B., Mosse, D., Melhem, R., & Craven, M. "Collaborative Compiler-OS Power Management for Time-Sensitive Applications", Transactions on Embedded Computing Systems, Vol. 5, No. 1, pp.82-115, February 2006.

[6] Pallipadi, V. "Enhanced Intel Speedstep Technology and Demand-based Switching on Linux", Intel Software Network, (retrieved 4/21/08 http://softwarecommunity.intel.com/articles/eng/1611.htm), October 2007.

[7] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (Chicago, IL, USA), June 2005.

[8] Pouwelse J., Langendoen, K., & Sips, H. J. "Application-Directed Voltage Scaling", Invited Paper, IEEE Transactions on Very Large Scale integration (TVLSI), September 2002.

[9] Holland, J.H., "Adaptation in Natural and Artificial Systems: An Introductory Analysis with Application to Biology, Control, and Artificial Intelligence", University of Michigan Press, 1975.

[10] De Jong, K.A, "Analysis of the Behavior of a Class of Genetic Adaptive Systems", PhD Dissertation, University of Michigan, 1975.

[11] Goldberg, D.E., and Deb, K., "A comparative analysis of selection schemes used in genetic algorithms" from "Foundations of Genetic Algorithms", Rawlins, G.J.E (ed), pp. 69-93, 1991.

[12] Eshelman, L.J., and Schaffer, J.D., "Preventing premature convergence in Genetic Algorithms by preventing incest", Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, pp.115-122, 1991.

Appendix A.1 – Definition of Raw Event Data Structure

```
typedef struct RawEvent {

  /**
   * Thread identifier
   */
  MachineAddress mThreadId;

  /**
   * Flags for type of block
   */
  EventFlags mFlags;

  /**
   * Address of call site.
   */
  MachineAddress mCallSiteAddress;

  /**
   * Address of function/method being called.
   */
  MachineAddress mCallToSiteAddress;

  /**
   * Cycle count read from performance counter register.  The ratio of time and
   * cycle count is directly proportional to CPU load w.r.t. concurrent threads.
   */
  uint64_t mCycleCount;

  /*
   * Time stamps read from on-chip time stamp counter
   */
  uint64_t mTimeStamp;
}
RawEvent;
```

Appendix A.2 – Example GA Configuration File

The following GA configuration data is taken from the float.exe gate test. Configuration values may have changed across gate tests.

```
Procs: 8
Freqs: 1853000000 1587000000
Timeslice: 18530000
Core sets:
        0 1 2 3
        4 5 6 7
Cache share cores:
        0 => 0
        1 => 0
        2 => 2
        3 => 2
        4 => 4
        5 => 4
        6 => 6
        7 => 6
Freq change penalty: 7246156
Instrumentation penalty: 161936540
False share penalty: 65000
Move penalty weight: 1000
Penalty matrix:
        0       386461  563589  579692  872597  872597  872597  872597
        386461  0       563589  579692  872597  872597  872597  872597
        563589  579692  0       386461  872597  872597  872597  872597
        563589  579692  386461  0       872597  872597  872597  872597
        872597  872597  872597  872597  0       386461  563589  579692
        872597  872597  872597  872597  386461  0       563589  579692
        872597  872597  872597  872597  563589  579692  0       386461
        872597  872597  872597  872597  563589  579692  386461  0
Watts per frequency:
        1587000000 => 7 1853000000 => 8.16
Min iterations: 2000
Coverge test count: 50
Iters per output: 50
Prob repro: 0.18
Prob cross: 0.8
Pop size: 100
Tourn size: 2
Gene type: 2
Time type: 0
Colocate thresh: 100
Min block cycles: 1000
```

Appendix A.3 – Notes

[i] Herein we use the term *platform* to mean a set of processing elements (e.g., multiple processors each with multiple cores).
[ii] Current Linux distributions, including Debian R4.0, contain two forms of the POSIX pthread libraries. One form uses processes to simulate threads, whilst the other uses native system threads (known as the Native Posix Thread Libraries, NPTL). To our knowledge, the NPTL implementation must be used in order to support control of true thread affinities.
[iii] Design optimization performed on a 1/10 snapshot of the total execution.